

Aufgabe 1

Ändern Sie die Sprache **WHILE** ab, indem Sie anstelle des atomaren Ausdrucks `read` Anweisungen der Form `read I` zulassen. Die Semantik dieser Anweisung lautet informell: Die Ausführung von `read I` bewirkt eine Zuweisung des nächsten Eingabewertes an die Variable `I` und eine Verkürzung der Eingabedatei um das erste Element.

Formalisieren Sie die Semantik von `read I` denotationell.

Tobi: Idee von Flo etwas weiter gedacht:

$$\mathcal{C}[\underline{read\ I}](s, e, a) = \begin{cases} (S[n/I], e', a), & \text{falls } \mathcal{T}[\underline{read}] s\ e = (n, e') \\ \text{Fehler}, & \text{sonst} \end{cases}$$

Soll heißen: der nächste gelesene Eingabewert (n) ersetzt (I) im Speicher. (e') ist ein um den ersten Wert kürzeres (e).

Aufgabe 2

Erweitern Sie die Sprache **WHILE** um Anweisungen der Form

`for I := T1 to T2 do C`

. Formalisieren Sie die Semantik dieser Anweisungen denotationell.

Diese for-Schleife lässt einem viel Raum zur Interpretation. Zum Beispiel trifft der Ausdruck $I := T_1 \text{ to } T_2$ keine Aussage darüber, wie T_1 und T_2 miteinander verglichen werden. Ebenfalls ist unklar, ob T_1 nach, vor, oder während der Ausführung von C inkrementiert, dekrementiert, o. ä., oder zugewiesen wird.

Wir gehen davon aus, das folgende denotationelle Semantik - angelehnt an die While-Definition - ausreichend ist.

$$\mathcal{C}[\underline{for\ I := T_1\ to\ T_2\ do\ C}](s, e, a) = \begin{cases} \mathcal{C}[C; \underline{for\ I\ to\ T_2\ do\ C}](s, e', a), & \text{falls} \\ \quad \mathcal{B}[\underline{T_1\ BOP\ T_2}] s\ e = (\underline{false}, e') \\ (s, e', a), & \text{falls } \mathcal{B}[\underline{T_1\ BOP\ T_2}] s\ e = (\underline{true}, e') \\ \text{Fehler}, & \text{sonst} \end{cases}$$

Anmerkung: Die erste Zuweisung von $I := T_1$ macht das Gesamte Konstrukt eigentlich noch komplizierter. Besser wäre eine Semantik mit einer weiteren Fallunterscheidung. Man hätte dann folgende Fälle: erste Ausführung, alle weiteren Ausführungen bei denen die Auswertung false ergibt, den Fall wo die logische Operation true ergibt und der Fehlerfall.

Aufgabe 3

Erweitern Sie die Sprache **WHILE** um den atomaren booleschen Term `eof`. Die informelle Semantik von `eof` lautet: `eof` ist wahr gdw die Eingabe leer ist.

Formalisieren Sie die Semantik von `eof` denotationell.

$$\mathcal{B}[\underline{eof}]z = \begin{cases} (\text{wahr}, z') & \text{falls } e = \epsilon \\ (\text{falsch}, z') & \text{sonst} \end{cases}$$

Aufgabe 4

Programmieren Sie in **WHILE** (einschließlich `eof`) einen Algorithmus zur Berechnung der Summe aller Eingabewerte. Beweisen Sie die Korrektheit Ihres Programms anhand der denotationellen Semantik. Diskutieren Sie die Problematik beim Fehlen von `eof`.

Programm in WHILE:

```
1 zahl := 0
2 while ¬ eof do zahl := zahl + read
```

Diskussion:

Ohne `eof` bräuchte das Programm für das Einlesen der kompletten Datei mittels einer Schleife eine Fehlerbehandlung (Exception-Handling), um eine leere Eingabe abzufangen, da das Programm sonst abstürzen würde (`read` wirft einen Fehler bei leerer Eingabe).